Scheduling an Overloaded Real-Time System *

Shyh-In Hwang
Computer Engineering and Science Department
Yuan-Ze Institute of Technology, Chung-Li, Taiwan 32026
Chia-Mei Chen
Citicorp Global Information Network, Citicorp
1900 Campus Commons Dr., Reston, VA 22091
Ashok K. Agrawala
Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland, College Park, MD 20742

The real-time systems differ from the conventional systems in that every task in the real-time system has a timing constraint. Failure to execute the tasks under the timing constraints may result in fatal errors. Sometimes, it may be impossible to execute all the tasks in the task set under their timing constraints. Considering a system with limited resources, one solution to handle the overload problem is to reject some of the tasks in order to generate a feasible schedule for the rest. In this paper, we consider the problem of scheduling a set of tasks without preemption in which each task is assigned criticality and weight. The goal is to generate an optimal schedule such that all of the critical tasks are scheduled and then the non-critical tasks are included so that the weight of rejected non-critical tasks is minimized. We consider the problem of finding the optimal schedule in two steps. First, we select a permutation sequence of the task set. Secondly, a pseudo-polynomial algorithm is proposed to generate an optimal schedule for the permutation sequence. If the global optimal is desired, all permutation sequences have to be considered. Instead, we propose to incorporate the simulated annealing technique to deal with the large search space. Our experimental results show that our algorithm is able to generate near optimal schedules for the task sets in most cases while considering only a limited number of permutations.

1 Introduction

Real-time computer systems are essential for all embedded applications, such as robot control, flight control, and medical instrumentation. In such systems, the computer is required to support the execution of applications in which the timing constraints of the tasks are specified by the physical system being controlled. The correctness of the system depends on the temporal correctness as well as the functional correctness of the tasks. Failure to satisfy the timing constraints can incur fatal errors. How to schedule the tasks so that their timing constraints are met is crucial to the proper operation of a real-time system.

In this paper, we consider the problem of creating a schedule for a set of tasks such that all critical tasks are scheduled, and then, among the non-critical tasks we select those which can be scheduled feasibly while maximizing the sum of the weights of selected non-critical tasks. As all systems have finite resources, their ability to execute a set of tasks while meeting the temporal requirements is limited. Clearly, overload conditions may arise if more tasks have to be processed than the available set of resources can handle. Under such overload conditions, we have two choices. We may augment the resources available, or reject some tasks (or both). Another permissible solution to this problem is to reject some of the tasks in order to generate a feasible schedule for the rest. Once a task is accepted by the system, the system should be able to finish it under its timing constraint.

For an overloaded system, how to select tasks for rejection on the basis of their importance becomes a significant issue. When the tasks have equal weight, an optimal schedule can be defined to be one in which the number of rejected tasks is minimized. In our previous study [3], we used a *super sequence* based scheduling algorithm to compute the optimal schedule for the tasks. In this paper, the criticality of the tasks are taken into consideration. Basically, if a task can not meet its deadline, it is rejected so that the CPU time would not be wasted. Secondly, we would like to schedule tasks such that the less important tasks may be rejected in favor of the more important tasks. We classify tasks into two categories: *critical* and *non-critical*. The critical tasks are crucial to the system such that they must not be rejected. The non-critical tasks are given weights to reflect their importance, and are allowed to be rejected. A schedule is *feasible* if all critical tasks in the task set are accepted and are guaranteed to meet their timing constraints. If there exists no feasible schedule for the task set, the task set is considered infeasible. The *loss* of a schedule is defined to be the sum of the weights of

*This work has been supported in part by Honeywell under N00014-91-C-0195 and Army/Phillips under DASG-60-92-C-0055, when the first author was studying in Computer Science Department, University of Maryland. The views, opinions, and/or findings contained in this report are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of Honeywell or Army/Phillips.

0-7803-3255-5/96 $5.00 © 1996 IEEE

the rejected non-critical tasks. A schedule is *optimal* if it is feasible and the loss of the schedule is minimum.

We first propose a Permutation Scheduling Algorithm (PSA) to generate an optimal schedule for a permutation. When it comes to scheduling a task set of n tasks, in the worst case there might be up to $n!$ permutations to consider. We propose a Set Scheduling Algorithm (SSA) which incorporates the *simulated annealing* technique [6] to deal with the large search space of permutations.

In the following section, we define the scheduling problem. In section 3, we present the idea about how to schedule a permutation. In section 4, we incorporate the technique of simulated annealing and discuss how to schedule a task set. In section 5, the results of our experiments are presented, which is followed by our conclusion.

2 The Problem

A task set is represented as $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$. A task τ_i can be characterized as a record of (r_i, c_i, d_i, w_i), representing the ready time, computation time, deadline, and criticality of the ith task. Time is expressed as a real number. A task can not be started before its ready time. Once started, the task must use the processor without preemption for c_i time units, and be finished by its deadline. If a task is very important for the system such that rejection of the task is not allowed, w_i is set to be CRITICAL. Otherwise, w_i is assigned an integral value to indicate its importance, and is subject to rejection if necessary. A *permutation sequence*, or simply abbreviated to a *permutation*, is an ordered sequence of tasks in the task set. Scheduling is a process of binding starting times to the tasks such that each task executes according to the schedule. Note that a non-preemptive schedule on a single processor implies a sequence for the execution of tasks. For the convenience of our discussion, we hereafter use a sequence to represent the schedule in the context. A permutation is denoted by $\mu = \langle \tau_1, \ldots, \tau_n \rangle$, where τ_i is the ith task in the permutation. A prefix of a permutation is denoted by $\mu_k = \langle \tau_1, \ldots, \tau_k \rangle$.

To schedule a task set, we need to take into consideration the possible permutations in the task set. We first consider an algorithm for scheduling a permutation. The finish time of a schedule is the finish time of the last task in the schedule. Let $S_k(t)$ denote a schedule of μ_k with finish time no more than t. We use $W(S_k(t))$ to represent the weight of $S_k(t)$, which is the sum of the weights of non-critical tasks in the schedule. A feasible schedule of μ_k is defined as follows:

Definition: $S_k(t)$, $1 \leq k \leq n$, is a *feasible* schedule of μ_k at t, if and only if:

1. $S_k(t)$ is a subsequence of μ_k,
2. the finish time of $S_k(t)$ is less than or equal to t, and
3. all critical tasks in μ_k are included in $S_k(t)$.

An optimal schedule of μ_k is defined as follows:

Definition: $\sigma_k(t)$ is an *optimal* schedule of μ_k at t, if and only if:

1. $\sigma_k(t)$ is a feasible schedule of μ_k, and
2. for any feasible schedule $S_k(t)$ of μ_k, $W(\sigma_k(t)) \geq W(S_k(t))$.

In other words, an optimal schedule is a feasible schedule with minimum loss. There are possibly more than one optimal schedules for μ_k with finish time less than or equal to t. We donote by $\Sigma_k(t)$ the set of all of the optimal schedules for μ_k at t. Hence, if $S_k(t) \in \Sigma_k(t)$, $S_k(t)$ is an optimal schedule for μ_k at t. The scheduling problem considered here is NP-complete.

3 Scheduling a Permutation

The methodology of finding an optimal schedule for the task set is presented in Figure 1. Clearly, to find the optimal schedule for the task set, all possible permutations have to be considered.

Loop 1: Choose a permutation μ of Γ
  Loop 2: for μ_k, $k = 1, 2, \ldots, n$
    Loop 3: compute $\sigma_k(t)$

Figure 1: Methodology

3.1 Computing $\sigma_k(t)$

We use dynamic programming to compute $\sigma_k(t)$ based on $\sigma_{k-1}(t')$, with $t' \leq t$.

If τ_k is a critical task, it is not difficult to see that

$$\sigma_k(t) = \sigma_{k-1}(t - c_k) \oplus \tau_k, \tag{1}$$

where $\oplus$ means concatenation of the sequence and the task. If τ_k is non-critical, our concern is to obtain as large a weight for the schedule as possible, while the critical tasks accepted previously must be kept in the schedule. Hence,

$$\sigma_k(t) = \begin{cases} \sigma_{k-1}(t - c_k) \oplus \tau_k & \text{or} \\ \sigma_{k-1}(t) \end{cases} \tag{2}$$

The chosen schedule has to be feasible, and has a weight more than or equal to the other.

3.2 Definition of Scheduling Points

In general, there exist a number of subranges in each of which the schedules are exactly identical [2]. We only need to compute the schedules at the time instants which delimit the subranges. We call these time instants *scheduling points*. The scheduling points can be determined by the timing characteristics of the tasks.

We denote the jth scheduling point for μ_k by $\lambda_{k,j}$, and call j the *index* of $\lambda_{k,j}$. Hence, $\sigma_k(\lambda_{k,j})$ denotes an optimal schedule for μ_k at the scheduling point $\lambda_{k,j}$. Let v_k be the total number of scheduling points at which we need to schedule μ_k.

For simplicity, λ_k denotes the set of $\lambda_{k,1}, \lambda_{k,2}, \ldots, \lambda_{k,v_k}$, and σ_k the set of $\sigma_k(\lambda_{k,1}), \sigma_k(\lambda_{k,2}), \ldots, \sigma_k(\lambda_{k,v_k})$. The scheduling points are defined as follows.

Definition: The set of scheduling points, λ_k, is *complete* if and only if:

1. for any $t < \lambda_{k,1}$, $\Sigma_k(t)$ is empty,
2. for any $\lambda_{k,j} \leq t < \lambda_{k,j+1}$, for $j = 1, \ldots, v_k - 1$, $\sigma_k(\lambda_{k,j}) \in \Sigma_k(t)$, and
3. for any $t \geq \lambda_{k,v_k}$, $\sigma_k(\lambda_{k,v_k}) \in \Sigma_k(t)$.

Note that $\Sigma_k(t)$ being empty means that there is no feasible schedule with finish time less than or equal to t. And also remember that $\sigma_k(\lambda_{k,j}) \in \Sigma_k(t)$ means that $\sigma_k(\lambda_{k,j})$ is an optimal schedule for μ_k at t. The completeness of scheduling points indicates that all of the optimal schedules at the positive real time domain can be represented by the optimal schedules computed at the scheduling points. In addition, the set of scheduling points, λ_k, is *minimum*, if and only if $W(\sigma_k(\lambda_{k,j})) < W(\sigma_k(\lambda_{k,j+1}))$, for any $1 \leq j \leq v_k - 1$. This ensures that there does not exist any redundant scheduling point which, if removed, does not violate the completeness of the scheduling points. The sets of scheduling points that we will discuss are complete and minimum.

3.3 Deriving Scheduling Points

We define the lower bound $L_{k-1} \equiv \lambda_{k-1,1}$, and the upper bound $U_{k-1} \equiv \lambda_{k-1,v_{k-1}}$. In particular, L_{k-1} denotes the largest time instant such that there is no feasible schedule for μ_{k-1} with finish time less than L_{k-1}. U_{k-1} denotes the least time instant such that the optimal schedule for μ_{k-1} with finish time greater than U_{k-1} can be $\sigma_{k-1}(\lambda_{k-1,v_{k-1}})$.

The six possible temporal relations in Equations 3–8 can be used to determine λ_k.

$$d_k - c_k < L_{k-1} \leq U_{k-1} \tag{3}$$

$$r_k \leq L_{k-1} \leq d_k - c_k < U_{k-1} \tag{4}$$

$$L_{k-1} < r_k \leq d_k - c_k < U_{k-1} \tag{5}$$

$$r_k \leq L_{k-1} \leq U_{k-1} \leq d_k - c_k \tag{6}$$

$$L_{k-1} < r_k \leq U_{k-1} \leq d_k - c_k \tag{7}$$

$$L_{k-1} \leq U_{k-1} < r_k \tag{8}$$

The temporal relations are illustrated in Figure 2, and can be summarized in three cases.

3.3.1 τ_k is Critical

The task τ_k must be the last task in any feasible schedule of μ_k. Remember that $\sigma_k(t)$ can be computed by Equation 1. In the following, we discuss how to derive the scheduling points for the three cases.

Case 1 $d_k - c_k < L_{k-1}$: μ is not feasible. Remember that there exists no feasible schedule for μ_k with finish time less than L_{k-1}, due to the completeness of scheduling points, and that $d_k - c_k$ is the latest start time for τ_k. Hence, μ_k is not feasible, and thus the whole permutation, μ, is not feasible.

Case 2 ($r_k \leq L_{k-1} \leq d_k - c_k$) or ($L_{k-1} < r_k \leq U_{k-1}$) : The scheduling points for μ_k is the set of $\lambda_{k-1,j} + c_k$, $j = 1, \ldots, v_{k-1}$, subject to the constraints that τ_k must start after r_k, and finish before d_k. Specifically, λ_k can be derived by Equations 9 and 10.

$$\lambda_{k,1} = max(\lambda_{k-1,1} + c_k, r_k + c_k) \tag{9}$$

Let J_{min} and J_{max} denote the smallest and the largest integers of j satisfying $\lambda_{k,1} < \lambda_{k-1,j} + c_k \leq d_k$. The rest of the scheduling points can be computed by

$$\lambda_{k,i} = \lambda_{k-1,j} + c_k, \tag{10}$$

where $J_{min} \leq j \leq J_{max}$ and $i = j - J_{min} + 2$.

Case 3 $U_{k-1} < r_k$: there is only one scheduling point. Since r_k is the earliest start time for τ_k, the only scheduling point is $r_k + c_k$.

3.3.2 τ_k is Non-critical

Remember that $\sigma_k(t)$ can be computed by Equation 2. The non-critical task τ_k is not necessarily included in the schedule for μ_k. Whether to include τ_k or not depends on how much weight may be gained by including τ_k.

Case 1 $d_k - c_k < L_{k-1}$: do nothing. The latest start time of τ_k is less than the lower bound, L_{k-1}; hence, τ_k can not be included in any feasible schedule. The scheduling points and schedules for μ_{k-1} remain the same as the scheduling points and schedules for μ_k. In our implementation, to save time and space, λ_{k-1} and λ_k use the same memory spaces; also, σ_{k-1} and σ_k use the same memory spaces. So now $\lambda_k = \lambda_{k-1}$ and $\sigma_k = \sigma_{k-1}$.

Case 2 ($r_k \leq L_{k-1} \leq d_k - c_k$) or ($L_{k-1} < r_k \leq U_{k-1}$) : If τ_k is included, the new *possible* scheduling points for μ_k is the set of $\lambda_{k-1,j} + c_k$, $j = 1, \ldots, v_{k-1}$, subject to the constraints that τ_k must start after r_k, and finish before d_k. Specifically, the new possible scheduling points , λ'_k, can be derived by Equations 11 and 12.

$$\lambda'_{k,1} = max(\lambda_{k-1,1} + c_k, r_k + c_k) \tag{11}$$

Let J_{min} and J_{max} denote the smallest and the largest integers of j satisfying $\lambda'_{k,1} < \lambda_{k-1,j} + c_k \leq d_k$. The rest of the scheduling points are

$$\lambda'_{k,i} = \lambda_{k-1,j} + c_k, \tag{12}$$

where $J_{min} \leq j \leq J_{max}$ and $i = j - J_{min} + 2$. If τ_k is not included, the scheduling points for μ_k are the old ones for μ_{k-1}; i.e.,

$$\lambda_{k-1,j},\ j = 1, \ldots, v_{k-1}. \tag{13}$$

It is worth mentioning that some optimal schedules may include τ_k, and some may not. The scheduling points, λ_k, can be derived by the following two steps.

1. Merge and sort the two arrays of scheduling points, λ'_k and λ_{k-1}, in Equations 11–13.

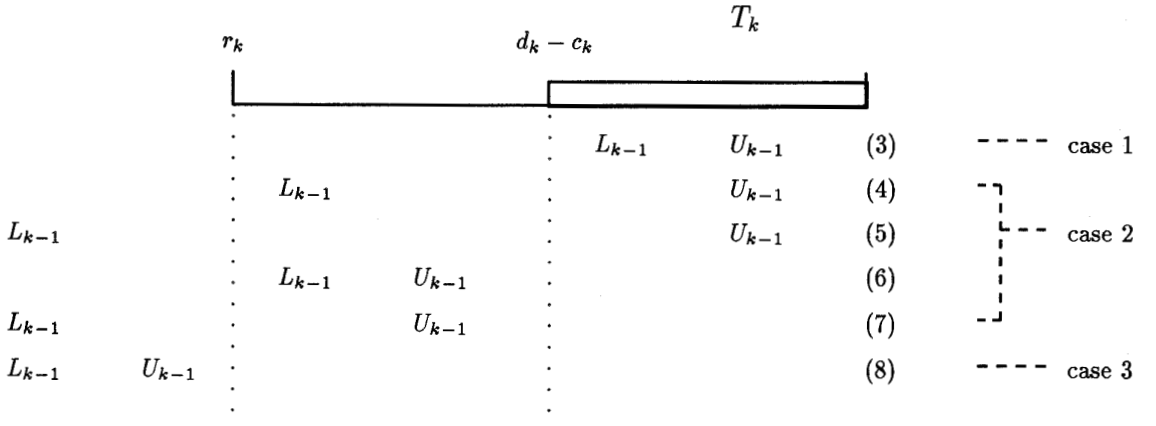

Figure 2: Temporal relations

2. The resultant array of scheduling points should follow the rule that the weights of the optimal schedules at the scheduling points should be strictly increasing. We remove any scheduling point that has a smaller weight than that of its preceding scheduling point in the array.

Case 3 $U_{k-1} < r_k$: add one more scheduling point. The earliest start time of τ_k is greater than the upper bound, U_{k-1}; hence, the new scheduling point is $r_k + c_k$. The weight of the optimal schedule computed at this scheduling point is $W(\sigma_{k-1}(\lambda_{k-1,v_{k-1}})) + w_k$, which is larger than $W(\sigma_{k-1}(\lambda_{k-1,v_{k-1}}))$. So this scheduling point must be included to make the set of scheduling points for μ_k complete. Note again that the scheduling points and schedules for μ_{k-1} remain unchanged as the scheduling points and schedules for μ_k; i.e., $\lambda_{k,j} = \lambda_{k-1,j}$ and $\sigma_k(\lambda_{k,j}) = \sigma_{k-1}(\lambda_{k-1,j})$, for $j = 1, \ldots, v_{k-1}$. However, $\lambda_{k,v_k} = r_k + c_k$ and $\sigma_k(\lambda_{k,v_k}) = \sigma_{k-1}(\lambda_{k-1,v_{k-1}}) \oplus \tau_k$, where $v_k = v_{k-1} + 1$.

Note that the sets of scheduling points derived in the three cases are complete and minimum [2].

3.4 The Permutation Scheduling Algorithm

Algorithm PSA:
Input: a permutation sequence $\mu = \langle \tau_1, \tau_2, \ldots, \tau_n \rangle$
Output: an optimal schedule $\sigma_n(\lambda_{n,v_n})$

Initialization: $v_0 = 1$; $\lambda_{0,1} = 0$; $\sigma_0(\lambda_{0,1}) = \langle\rangle$;
$W(\sigma_0(\lambda_{0,1})) = 0$
for $k = 1$ to n
when τ_k is critical
case 1 $(d_k - c_k < L_{k-1})$
exit
case 2 $(r_k \le L_{k-1} \le d_k - c_k)$ or $(L_{k-1} < r_k \le U_{k-1})$:
$\lambda_{k,1} = max(\lambda_{k-1,1} + c_k, r_k + c_k)$
$j = 1$ if $\lambda_{k-1,1} > r_k$; otherwise, j is the greatest integer such that $\lambda_{k-1,j} \le r_k$
$\sigma_k(\lambda_{k,1}) = \sigma_{k-1}(\lambda_{k-1,j}) \oplus \tau_k$
$W(\sigma_k(\lambda_{k,1})) = W(\sigma_{k-1}(\lambda_{k-1,j}))$
Loop: $j = J_{min}$ to J_{max}
$i = j - J_{min} + 2$
$\lambda_{k,i} = \lambda_{k-1,j} + c_k$
$\sigma_k(\lambda_{k,i}) = \sigma_{k-1}(\lambda_{k-1,j}) \oplus \tau_k$
$W(\sigma_k(\lambda_{k,i})) = W(\sigma_{k-1}(\lambda_{k-1,j}))$
$v_k = J_{max} - J_{min} + 2$
case 3 $(U_{k-1} < r_k)$: (only one scheduling point)
$\lambda_{k,1} = r_k + c_k$
$\sigma_k(\lambda_{k,1}) = \sigma_{k-1}(\lambda_{k-1,v_{k-1}}) \oplus \tau_k$
$W(\sigma_k(\lambda_{k,1})) = W(\sigma_{k-1}(\lambda_{k-1,v_{k-1}}))$
$v_k = 1$

when τ_k is non-critical
case 1 $(d_k - c_k < L_{k-1})$
/* $\lambda_k = \lambda_{k-1}$ and $\sigma_k = \sigma_{k-1}$ */
case 2 $(r_k \le L_{k-1} \le d_k - c_k)$ or $(L_{k-1} < r_k \le U_{k-1})$:
$\lambda'_{k,1} = max(\lambda_{k-1,1} + c_k, r_k + c_k)$
$j = 1$ if $\lambda_{k-1,1} > r_k$; otherwise, j is the greatest integer such that $\lambda_{k-1,j} \le r_k$
$\sigma'_k(\lambda'_{k,1}) = \sigma_{k-1}(\lambda_{k-1,j}) \oplus \tau_k$
$W(\sigma'_k(\lambda'_{k,1})) = W(\sigma_{k-1}(\lambda_{k-1,j})) + w_k$
Loop: $j = J_{min}$ to J_{max}
$i = j - J_{min} + 2$
$\lambda'_{k,i} = \lambda_{k-1,j} + c_k$
$\sigma'_k(\lambda'_{k,i}) = \sigma_{k-1}(\lambda_{k-1,j}) \oplus \tau_k$
$W(\sigma'_k(\lambda'_{k,i})) = W(\sigma_{k-1}(\lambda_{k-1,j})) + w_k$
construct σ_k from σ_{k-1} and σ'_k by
1) merging and sorting λ_{k-1} and λ'_k into one array
2) making the weights of the resultant schedule strictly increasing.
case 3 $(U_{k-1} < r_k)$:
$v_k = v_{k-1} + 1$
$\lambda_{k,v_k} = r_k + c_k$
$\sigma_k(\lambda_{k,v_k}) = \sigma_{k-1}(\lambda_{k-1,v_{k-1}}) \oplus \tau_k$
$W(\sigma_k(\lambda_{k,v_k})) = W(\sigma_{k-1}(\lambda_{k-1,v_{k-1}})) + w_k$
endfor

4 Scheduling a Task Set

To find an optimal schedule for the task set, we may have to consider all possible $(n!)$ permutations. It is possible to reduce the search space by eliminating some infeasible permutations. For example, if $d_i < r_j$, there is no feasible schedule in which τ_i is placed after τ_j. Even after the reduction, the search space might still be too large. We propose to use *simulated annealing* [4, 6] technique, recognizing that while this technique reduces the search, it may yield sub-optimal results.

4.1 The Set Scheduling Algorithm (SSA)

A permutation is used to represent the configuration. If a permutation is ordered in an Earliest Deadline First (EDF) fashion, we call it an EDF permutation. The energy function can be expressed by a loss function:

$$loss = \sum weight\ of\ rejected\ noncritical\ tasks$$

A schedule is not acceptable if critical tasks are rejected. We may say that the loss of a rejected critical task is infinity. However, this kind of assignment makes it difficult to distinguish between a very bad schedule (e.g., a critical task is rejected) and even a worse schedule (more critical tasks are rejected). In general, the former schedule can be considered as an improvement over the latter one. If the loss incurred by a rejected critical task is assigned infinity, there is no way to tell which is better between the schedule in which one critical task is rejected and that in which three critical tasks are rejected. Hence, we assign a finite amount of loss to rejected critical tasks. The loss of a critical task must be large enough such that the scheduler will not reject a critical task to accommodate a number of non-critical tasks.

The neighbor function may be obtained using one of the following two methods. In the first, simple method, we randomly select one task from those rejected. This task is inserted in a randomly chosen location within a specified distance from its original location, where the *distance* is the number of tasks between two tasks in a permutation. The distance is used in this approach to control the degree of perturbation. In the second method, we try to identify the task which causes the largest loss of weight. As a simple approach, we attribute the rejection of a task to the task accepted prior to it. Then we choose the task which causes the largest loss of weight and insert it within a specified distance. The Set Scheduling Algorithm (SSA) is presented in Figure 3.

The initial temperature has to be large enough such that virtually all up jumps are allowed in the beginning of the annealing process. According to [6], the way to compute new temperature is that new temperature $= \alpha *$ current temperature, where $0 \leq \alpha \leq 1$. A step denotes an iteration in the inner loop in Figure 3, which is the process of scheduling a permutation and determining whether the permutation would become the current permutation. The thermal equilibrium can be reached if a certain number of down jumps or a certain

Algorithm SSA:

Begin
  choose initial temperature T
  choose edf permutation as the starting permutation, μ
  schedule μ by **PSA** and compute its energy, E
  loop
    loop
      compute neighbor permutation μ'
      schedule μ' by **PSA** and
        compute its energy, E'
      if $E' < E$ **then**
        making μ' the current permutation:
          $\mu \leftarrow \mu'$ and $E \leftarrow E'$
      else
        if $e^{\frac{E-E'}{T}} \geq$ random(0,1) **then**
          making μ' the current permutation:
            $\mu \leftarrow \mu'$ and $E \leftarrow E'$
        else
          μ remains as the current permutation
    until thermal equilibrium is reached
    compute new temperature: $T \leftarrow \alpha * T$
  until stopping condition is reached
End

Figure 3: Set Scheduling Algorithm

number of total steps has been observed; and the freezing point, or the stopping condition, can be reached if no further down jump has been observed in a certain number of steps [4, 6].

5 Experiment Result

Experiments are conducted to study the performance of SSA based on:

- scheduling ability = (number of times that the algorithm generates a feasible schedule) / (number of times that there does exist a feasible schedule for the task set)
- loss ratio = (loss of the schedule generated by SSA - loss of an optimal schedule) / (total weight of accepted noncritical tasks of an optimal schedule)
- iterations = number of permutations that the simulated annealing algorithm goes through to obtain the sub-optimal schedule

We start with an EDF permutation. To study how good the result would be by using PSA to schedule the EDF permutation, the scheduling ability and loss ratio for the EDF permutation are computed as well. In our experiments, a task set consists of 100 tasks. The number of permutations in such a task set is $100! \approx 9.33 * 10^{157}$. To study how good the output of SSA is compared to an optimal schedule, it

parameters	*value*	*type*
window length	mean_Wl = 20.0	truncated normal distribution
computation time	mean_C = $\frac{mean\_Wl}{3}$	truncated normal distribution
load	20%, 40%, 60%, 80%	constants
criticality ratio	25%, 50%, 75%	constants
weight	low_W=1, high_W=50	discrete uniform distribution

Figure 4: Parameters of the experiments

is rather impractical to go through such a great number of permutations for a task set to derive the optimal schedule and its minimum loss for comparison. Instead, we choose to make up a task set such that the task set is feasible and the loss of its optimal schedule is 0. Although the SSA algorithm is primarily designed for an overloaded system, we apply SSA to such task sets for measuring the performance. The parameters are shown in Figure 4.

The mean of window length, mean_Wl, is set to be 20 time units. The load is the ratio of total computation time to the largest deadline, D, in the task set. Hence, the load indicates the difficulty of scheduling the task set. The mean of computation time, mean_C, is one third of the mean of window length, which allows the windows among tasks to overlap to some extent. How much the windows overlap partially depends on the load. If the load is high, the windows are congested together, and thus the overlapping is high. We expect some containing relations between tasks to occur and thus increase the difficulty for scheduling. Note that, without containing relations, scheduling the task set would be straightforward. The standard deviations of window length and computation time are set to be their means, respectively. Criticality ratio indicates the percentage of the critical tasks in the task set. It is set to be 25%, 50%, and 75%. The higher the criticality ratio, the more difficult it is to generate a feasible schedule for the task set. On the other hand, although it is easier to come up with a feasible schedule when the criticality ratio is low, the loss ratio may still be high. It may be necessary to go through many permutations before an acceptable loss ratio is reached. In our experiments, the acceptable loss ratio is set to be 0%, which means that SSA will keep trying different permutations until either the loss ratio is 0 or the stopping condition is reached, in which SSA fails to find an optimal schedule. Note that a big energy (loss), 1000, is incurred for a rejected critical task. Hence, for an infeasible schedule, the loss ratio may well be more than 100%. The weight of a non-critical task is an integer ranging from low_W=1 to high_W=50, determined by a discrete uniform distribution function. For each individual experiment with different parameters, 200 task sets, each with 100 tasks, are generated for scheduling.

From Figure 5a, the scheduling ability of SSA is 98.5% when criticality ratio is 75% and load is 80%, and is 100% for other lower criticality ratios and loads. This is because the simulated annealing algorithm focuses on searching suitable neighbor permutations in such a way that the rejected critical tasks, if any, may be accepted.

As far as non-critical tasks are concerned, SSA can not guarantee the minimum loss. However, even in the worst case given in Figure 5b, the loss ratio is less than 10%.

The number of permutations to be searched in simulated annealing depends on the situations of energy jumps, the way of reducing temperature, and how we define thermal equilibrium and stopping conditions. How to set the parameters in simulated annealing differs a great deal from one application to another. We find that the following parameters are beneficial: initial temperature = 3000, $\alpha = 0.8$ (instead of 0.95 or even 0.99 suggested in other applications), the number of down jumps to obtain thermal equilibrium = 25, the number of total steps to obtain thermal equilibrium = 300, the number of steps with no further down jump to obtain the freezing point = 2000, which is also the stopping condition. The average number of permutations searched in simulates annealing is given in Figure 5c. If SSA can successfully generate a feasible schedule, the average number of permutations checked is no more than 4000 times. The number increases a little if SSA fails to find a feasible schedule, because in this case SSA does not stop until the freezing point is reached. Note that the average numbers of permutations are less than n^2, which can roughly give us the idea about the complexity of searching over the permutation space.

6 Conclusion

In this paper, we study the scheduling problem for a real-time system which is overloaded. A significant performance degradation may be observed in the system if the overload problem is not addressed properly [1]. As not all the tasks can be processed, the set of tasks selected for processing is crucial for the proper operation of an overloaded system. We assign to the tasks *criticalities and weights* on the basis of which the tasks are selected. The objective is to generate an optimal schedule for the task set such that all of the critical tasks are accepted, and then the loss of weights of non-critical tasks is minimum.

We present a two step process for generating a schedule. First, we develop a schedule for a permutation of tasks using a pseudo-polynomial algorithm. The concept of *scheduling points* is proposed for the algorithm. In order to find the optimal schedule for the task set, we have to consider all permutations. The simulated annealing technique is used to limit the search space while obtaining optimal or near optimal

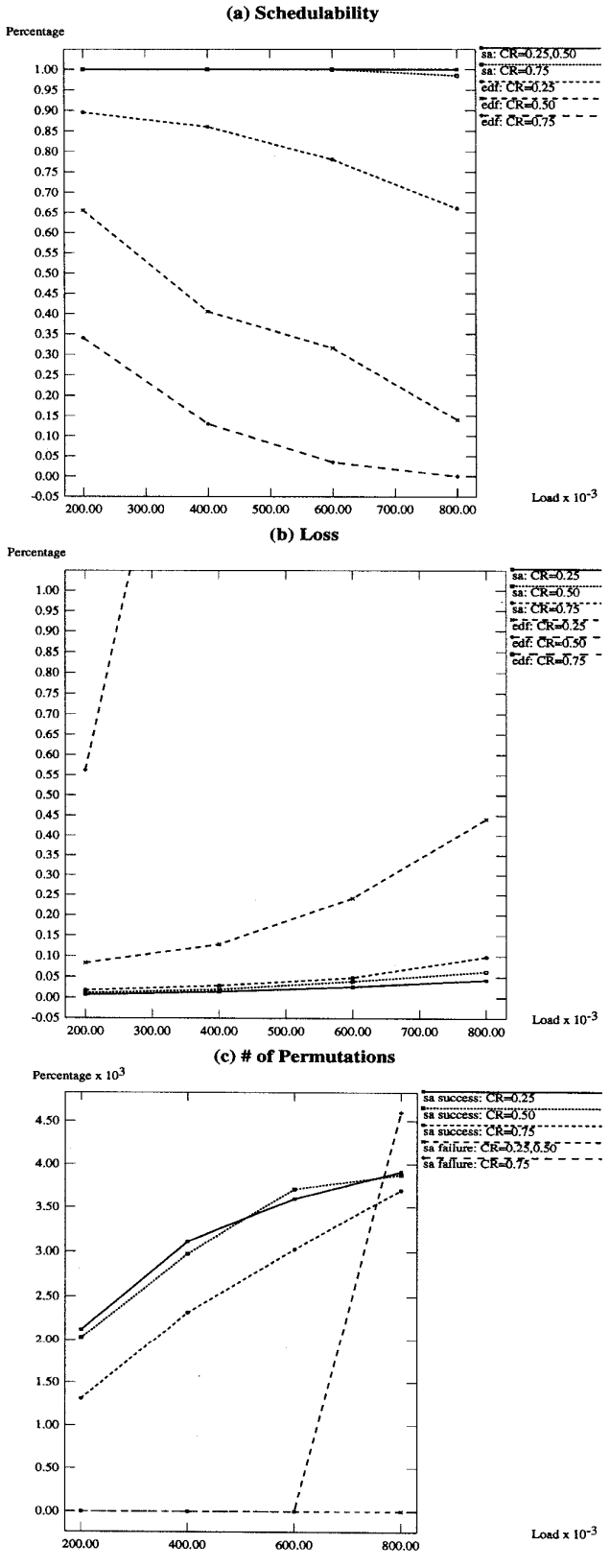

Figure 5: (a) Schedulability (b) Loss (c) #Permutations

results. Our experimental results indicate that the approach is very efficient.

The work presented in this paper can be easily extended to address the overload issue for periodic tasks. To schedule a set of periodic tasks with criticalities and weights, we can convert the periodic tasks in the time frame of the least common multiple of the task periods to aperiodic tasks. The schedule generated for the frame can be applied repeatedly for the subsequent time frames.

Our algorithm can also be applied to solving the problem of scheduling imprecise computations [5], in which a task is decomposed logically into a mandatory subtask, which must finish before the deadline, and an optional subtask, which may not finish. The goal is to find a schedule such that the mandatory subtasks can all be finished by their deadlines and the sum of the computation times of the unfinished optional subtasks is minimum. A schedule satisfies the *0/1 constraint* if every optional subtask is either completed or discarded [5]. We can solve this problem by using our algorithm by setting the mandatory subtasks to be critical, and the optional subtasks to be non-critical with weights equal to their computation times.

References

[1] J. R. Haritsa, M. Livny, and M. J. Carey. Earliest deadline scheduling for real-time database systems. In *IEEE Real-Time Systems Symposium*, Dec. 1991.

[2] S. I. Hwang, C. M. Chen, and A. K. Agrawala. Scheduling an overloaded real-time system. Technical Report CS-TR-3377, UMIACS-TR-94-128, Department of Computer Science, University of Maryland at College Park, Nov. 1994.

[3] S. I. Hwang, S. T. Cheng, and A. K. Agrawala. An optimal solution for scheduling real-time tasks with rejection. In *International Computer Symposium*, Dec. 1994.

[4] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science(220)*, pages 671–680, 1983.

[5] W. K. Shih, J. Liu, and J. Y. Chung. Fast algorithms for scheduling imprecise computations. In *IEEE Real-Time Systems Symposium*, pages 12–19, Dec. 1989.

[6] K. W. Tindell, A. Burns, and A. J. Wellings. Allocating hard real-time tasks: An np-hard problem made easy. *The Journal of Real-Time Systems*, 4(2):145–165, June 1992.